Modular Open Robots Simulation Engine: MORSE

Gilberto Echeverria and Nicolas Lassabe and Arnaud Degroote and Séverin Lemaignan

***Abstract*— This paper presents MORSE, a new open–source robotics simulator. MORSE provides several features of interest to robotics projects: it relies on a component-based architecture to simulate sensors, actuators and robots; it is flexible, able to specify simulations at variable levels of abstraction according to the systems being tested; it is capable of representing a large variety of heterogeneous robots and full 3D environments (aerial, ground, maritime); and it is designed to allow simulations of multiple robots systems. MORSE uses a "Software-in-the-Loop" philosophy, *i.e.* it gives the possibility to evaluate the algorithms embedded in the software architecture of the robot within which they are to be integrated. Still, MORSE is independent of any robot architecture or communication framework (middleware).**

MORSE is built on top of Blender, using its powerful features and extending its functionality through Python scripts. Simulations are executed on Blender's Game Engine mode, which provides a realistic graphical display of the simulated environments and allows exploiting the reputed Bullet physics engine. This paper presents the conception principles of the simulator and some use–case illustrations.

I. Introduction

Robotics systems are becoming highly complex and sophisticated, with an increasing number of hardware and software components. There is also an increasing variety of tasks involved in performing robotics experiments, which induces much time and resources for validation. The use of a simulator can ease the development and validation processes, allowing to verify the component integration and to evaluate their behaviour under different controlled circumstances. Roboticists are currently paying a lot of attention to the development of robotics simulation, as shown in recent workshops [1] and conferences [2].

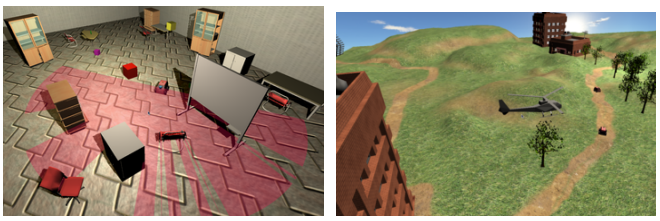
Simulation is cheaper in terms of time and human resources than experiments with real robots. It can also provide more flexibility, by allowing testing under conditions that would be unfeasible otherwise: a simulated environment can be significantly more complex and larger than a lab environment, and meanwhile ensure a perfect repeatability. Another advantage is the possibility of simulating multiple robots when the hardware may not be available.

In robotics, we can distinguish two main types of simulators. Those restricted to the validation of a specific kind of component, *e.g.* the processing of data provided by a given sensor or path planning for a particular kinematic system [3], [4], [5]; such simulators are highly specialised (*"Unitary Simulation"*). On the other hand, some applications require a more general simulator that can allow the evaluation of a robot *at the system level*, *i.e.* represent all aspects of a robot as a whole [6]. MORSE belongs to the latter kind of simulators, which are more versatile, flexible and reusable.

G. Echeverria is with RTRA STAE, 23 avenue Edouard Belin, 31400 Toulouse, France gechever@laas.fr

N. Lassabe is with ONERA Centre de Toulouse – DCSD, 2 avenue douard Belin, F-31055 Toulouse, France nicolas.lassabe@onera.fr

A. Degroote and S. Lemaignan are with CNRS, LAAS, 7 avenue du colonel Roche, F-31077 Toulouse, France and Université de Toulouse, UPS, INSA, INP, ISAE, LAAS, F-31077 Toulouse, France adegroot@laas.fr and slemaign@laas.fr



(a) A mobile robot in an indoor scene (b) A helicopter and two mobile ground robots outdoors

Fig. 1: Screenshots of the MORSE simulator.

This paper presents the architecture of the MORSE simulator, jointly developed at LAAS and ONERA. It is built on top of the Blender software and is intended as a general purpose, modular system simulation of multiple moving robots in any kind of environment (Fig. 1), and provides a library of configurable components that can be interconnected to create any robot configuration. The outline of the paper is as follows: the motivations and requirements for the development of another robotics simulator are stated in Section II. Section III presents the overall architecture of the simulator and its components. Section IV shows the current state of MORSE development, while planned future developments are presented in Section V. Finally Section VI provides a general discussion on the MORSE simulator.

II. Motivations – requirements

Various commercial robotics simulators are already available [7], [8], [9], [10], as well as open-source [11], where the most popular are *Player/Stage* [12] and *Gazebo* [13]. These projects are currently limited in terms of system simulation of robots in a realistic 3D environment. The MORSE simulator is an open-source application (BSD-3 clauses) that can be used in different contexts for the testing and verification of robotics systems as a whole, at a medium to high level of abstraction. It is not meant to replace dedicated simulators for very specific purposes. One of the main interests of creating a new simulator is making it reusable by researchers and engineers: MORSE is being developed as part of multiple projects, each with different restrictions and requirements,

but all related with multiple robot interaction and cooperation, as well as human–robot interaction. The projects require the representation of heterogeneous robots, each with their own set of capabilities. For this reason the design of MORSE must be completely modular, as in many other modern simulators. Another strong requirement for the simulator is that it must be capable of interacting with any middleware used in robotics, and not impose a format that others must adapt to. MORSE is designed to handle the simulation of several robots simultaneously, as a distributed application where the robotics software being evaluated can run on the same or a different computer as the simulation.

A. Simulation based on Blender

Blender [14] is an open source 3D modelling and rendering application whose main purpose is the creation of computer generated images and animations. Though it is not designed as a tool for simulation, it provides many features that facilitate the development of such an application. There exists already a community of robotics researchers who use Blender for some simulations [15], and there is a drive to improve on this functionality.

The most obvious advantage of using Blender is the high level of graphical detail that can be achieved in real time, thanks to the advanced modelling of meshes, and effects such as texturing, lighting and shaders. The visual aspect is important when simulating robotic vision, since the images captured in the virtual world can be realistic enough (see Fig. 1) to be processed with the same algorithms as real images. Blender also offers the capability of using several camera views to follow the evolution of the simulation, displaying a global view of the scenario, as well as views from each of the cameras on–board the various robots.

Blender provides the tools necessary to model robots and scenarios with as much detail as required. Furthermore, it also gives immediate access to the Bullet engine for physics simulation. The interface with the modelled objects is already integrated, and the physical properties of objects can be specified in control panels. These properties include the mass and friction of an object, its bounding box to detect collisions, its interaction with other objects and the force of gravity in the virtual world. The recent Blender 2.5 version incorporates the iTasC Inverse Kinematics solver [16], which permits the use of IK armatures in the Game Engine, useful for simulating robotic arms and humans.

The element in Blender that permits the development of an interactive simulation is the *Game Engine* (GE) mode. It provides the user with a flexible graphical interface (called the *Logic Bricks*) to script behaviour to objects in the scene, and to define variables (called *Logic Properties*) associated with the same objects. One of the Logic Bricks also permits the use of Python scripts which can interact with the Blender world though a dedicated API. Additional modules can also be programmed in Python or C/C++ using SWIG wrappers.

For all the advantages of using Blender, there is also the drawback of having to understand its (non-trivial) interface, as well as the additional computational overhead of using a software for a purpose different that expected.

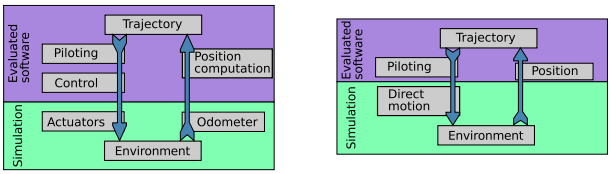


Fig. 2: Simulation of a trajectory following process at two different levels of abstraction. Left: low abstraction simulation, giving linear and angular velocities for the movement. Right: higher abstraction simulation, using a direct destination coordinate.

B. Simulation at different levels of abstraction

We aim at defining a simulation infrastructure that can be exploited to develop and validate a wide spectrum of robotics functionalities, ranging from the simplest ones, *e.g.* navigation in a flat environment, to complex scenarios that involve a fleet of heterogeneous robots, *e.g.* aero-terrestrial teams. For such purposes, the simulator should be able to work at *various levels of abstraction*. A realistic (non-abstract) simulation produces exactly the same data as the sensors of the actual robot, and accepts actuator commands as they are sent to the robot, whereas a more abstract simulation produces and/or accepts higher level data and commands. Letting the user the possibility to simulate a robot at various levels of abstraction is essential, as it allows him to specify the functions he wants to simulate and the ones he wants to evaluate. For instance, when evaluating a high level algorithm, it is not necessary to worry about lower level actions, and an abstract simulation is sufficient. Fig. 2 and 3 illustrate this point for actuators and sensors, respectively. They show two possible separations between the virtual environment and its data (*simulation*) and the robotics software to be tested (*evaluated software*).

C. "Software-in-the-Loop"

MORSE is designed to interact directly with the evaluated software exactly as it is, without the need of any modifications to the software. This philosophy takes after "Hardware-in-the-Loop" simulations, in which the evaluated components are run on the target hardware and interact with the simulator with the very same protocols than the ones of the actual robots sensors and actuators [17], in order to make the shift from simulations to actual experiments totally transparent. "Software-in-the-Loop" means that when linked with the simulator, the evaluated components are embedded in the same target architecture than the considered robot: MORSE can link with the evaluated software layer by the use of any middleware, as explained in the next section. MORSE is nevertheless *independent* of any middleware, and can be used in scenarios involving simultaneously robots with different software architectures: contrary to some simulation schemes (*e.g. [18]*), MORSE is not tied to any specific robot middleware or software architecture.

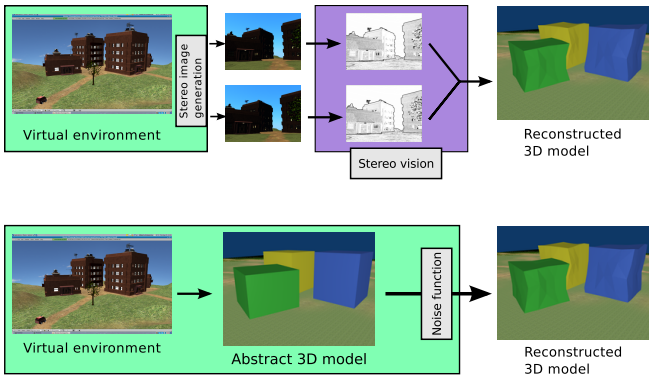


Fig. 3: Simulation of the perception of a 3D scene at two different levels of abstraction. Top: the simulation produces an image pair (as a stereo vision bench would) that is further processed "on–board the robot" to produce a 3D vector representation of the perceived scene. Bottom: a more abstract simulation directly produces the 3D representation directly from the Blender geometry. In this latter case, a "noise function" within the simulator reproduces the imperfect data generated by the stereo vision process.

III. Overall architecture

A. Modular design

MORSE, while dedicated to robotic simulation, relies on the Blender approach of file composition to build simulated scenes: one scene can reference a Blender object stored in other file. When the original object is updated, this is in turn reflected in all scenes that depend on this asset. MORSE makes use of this modular philosophy, providing a library of simple components that can be assembled together with others. Each MORSE component consists of a Python and a Blender file: The Python file defines an object class for the component type, with its state variables, data and logical behaviour (methods). All components extend from an abstract base `MorseObjectClass` that defines: the 3D position and orientation relative to the origin of the Blender world, a `local_data` dictionary with the information the component must share with other elements and a basic `default_action` method that individual subclasses must implement according to their functionality. Sensors and actuators have also a reference to the robot they are attached to, and the relative position/orientation with respect to it. The base class also provides lists of additional functions that can be added dynamically to the components during runtime to extend their basic functionality. The Blender file specifies the visual and physical properties of the object in the simulated world. The complexity of the component meshes can vary from a simple cube to a complete robotic arm of multiple segments. The physical properties include colour, texture, dynamic properties like friction or mass, and possibly other data like simplified bounding box models for collision detection. The Blender file holds as well the overall component logic (expressed with the so-called *Logic Bricks*) that binds Python methods to events generated in the Blender world. Every component has variables (`Logic Properties`) indicating its kind, and the Python file that provides its class and functionality.

There are currently three different kinds of robotics components defined in MORSE.

- Sensors: Recover data from the simulated world, emulating the functionality of the real sensors by using the logic functionality of the GE.
- Actuators: Produce actions upon the associated robots or components. In particular, actuators move the robots based on a given parameter: destination coordinates or linear/angular velocities. Other actuators can affect other components, such as the position of arms or pan-tilt units on robots.
- Robots: The platforms where sensors and actuators are mounted. They also define the physical properties (size, weight, friction, mobility, collision bound) of the virtual robot. Robotic arms are also included in this category. These are composed of several segments that can be articulated and that have special actuators.

Besides those, three other classes of components are available:

- Scenes: Modelled environments where the robot will interact during the simulation. MORSE provides some examples like a outdoor scene with trees and buildings or a furnished indoor room (see Fig. 1). MORSE scenes are simple Blender scenes and any previously modelled environment can be reused.
- Middlewares: Communication channels between the simulator and the *evaluated software* are set up through special Blender objects that take care of the bindings with the simulated sensors and actuators.
- Modifiers: Simulated sensors produce "perfect data", with very accurate measures taken from the virtual world. This is never the case in the real world, where some sort of noise is always present in the data. It is possible to add *Modifiers* that encapsulate functions to alter the data produced by the simulator, typically noise functions. These modifiers expose methods that are inserted in the data flow between the simulation and the *evaluated software*.

B. Integration with middlewares

Middlewares are an intermediate layer between different software systems that enables them to communicate and share data. Various middlewares are used in robotics, such as YARP [19], ROS [20], Pocolibs [21] and others [22], [23], [24]. While middlewares are designed to connect separate components, if an element is highly coupled with a given middleware, then it becomes difficult to reuse it in a different environment. For this reason, components should be designed to be middleware–independent.

The philosophy of MORSE is influenced by another software package used at LAAS: G^{en}oM 3 [25]. It is a tool to generate software modules that can be compiled with any middleware. This permits keeping the code of a component

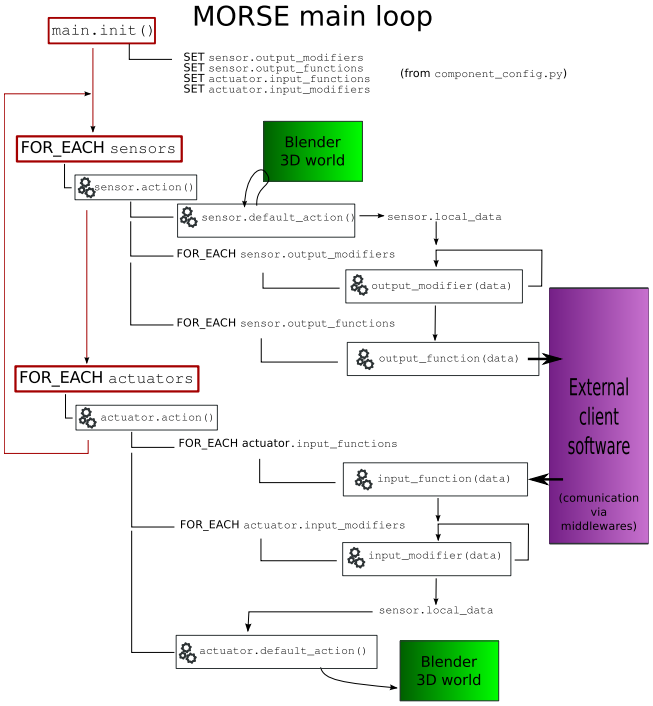


Fig. 4: The data flow between components of the simulator.

completely separated from the middleware, and only bound to it when it is necessary. Once a module has been generated, it can be connected to the rest of the robot software, and exchange data by means of data structures and requests.

MORSE components (sensors or actuators) store the data they use in the Python dictionary `local_data`. However, to be middleware independent, they do not provide any functionality destined to share their data with other programs outside of Blender. We implement a mechanism called *hook* to dynamically add middleware bindings when necessary. The mechanism consists on adding methods to the component instances during runtime, thanks to the dynamic nature of Python. These methods must access the `local_data` dictionary and prepare the data to be sent in the format required by the corresponding middleware. Fig. 4 shows the data flow of the sensor and actuator components using *hooks* to share data with external applications.

The binding of specific components and middlewares in a simulation scene is defined in a script called `component_config.py`, inside the Blender file for the scene. The script consists of a Python dictionary listing the components, the middleware and the method each will use (See example below). When the simulation is started, this file is read, and the methods listed are added to the class instance of the components.

```
component_mw = {
    "Motion_Controller": ["Socket", "read_message"],
    "GPS": ["Yarp", "post_message"],
    "Gyroscope": ["Text", "write_data"],
    "PTU": ["Pocolibs", "write_viam",
        "morse/middleware/pocolibs/sensors/viam"],
    "Sick": ["Yarp", "post_sick_data",
        "morse/middleware/yarp/sick"],                }
```

Thanks to this *hook* mechanism, MORSE can run several middlewares in parallel: one robot could be controlled with YARP with one of its sensors logged through a socket, while another robot is managed by a software running Pocolibs. MORSE middleware modules are considered as optional plug–ins. An instance of them is created if necessary, with the initialization routine specific to each. They also implement the basic data serialisation necessary to transfer information to external software. To simplify deployment, each middleware module provides a default, basic data serialization for the common data types. For more complex data (as is the case for images, arrays or other structures) an additional module must be included that will define the specific serialisation necessary for each data structure. In its current version, MORSE has support for YARP, Pocolibs, ROS, raw sockets and a dummy text-based input/output mechanism (useful for creating logs). Modifiers also make use of the *hook* mechanism to add noise or otherwise alter the data of each component. Currently there exists modifiers that can generate gaussian noise, and others that convert coordinates to UTM (Universal Global Mercator) used by GPS systems or transform the reference system of the X, Y and Z axis from ENU (East, North, Up) to NED (North, East, Down).

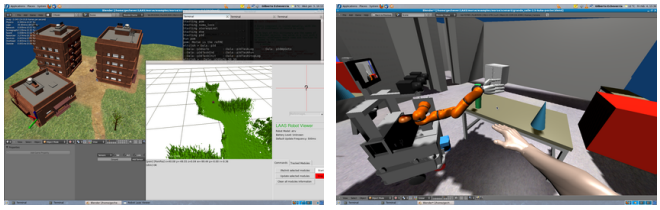
C. Construction of a simulation scenario

Building a complete simulated robot in MORSE involves the creation of a new Blender file, with the robot mesh and its components linked from external individual Blender files. The sensors and actuators must be set as children of the robot, so that they will move together and share data. Just as in real robots, the relative position of the sensors with respect to the robot frame is important, as it must be considered to correctly locate the data when they are generated.

When a virtual robot is complete, it is itself linked to a scene file with the virtual world in another Blender file. Linked components can be modified or duplicated inside the scene as necessary, using Blender's modelling tools. All environmental settings must be specified in this scenario file, such as the global coordinate system. The list of the bindings of middlewares with individual components is also defined in the scenario file, as the script `component_config.py`. The modules for the middlewares and modifiers to be used in the scene must also be linked into the scenario file.

IV. Current status

A number of components are already available for MORSE, which allows to rapidly produce test scenarios of specific cases. Robot platforms already modelled include: iRobot ATRV Unmanned Ground Vehicle (UGV), Yamaha RMAX Unmanned Aerial Vehicle (UAV) and a NeoBotix mobile platform. Robotic arms available are Mitsubishi PA-10 and KUKA LWR, both implemented with inverse kinematics. A user controlled human model is also available in MORSE and permits human robot interaction inside the simulated scenarios. Some of the sensors already implemented in MORSE are: Cameras, Gyroscope, GPS, Accelerometer, Thermometer, SICK laser and proximity detectors. The



(a) DTM generation from treatment of stereo camera images

(b) Objects on the table are tagged and identified without processing

Fig. 5: Camera sensors with different information abstraction.

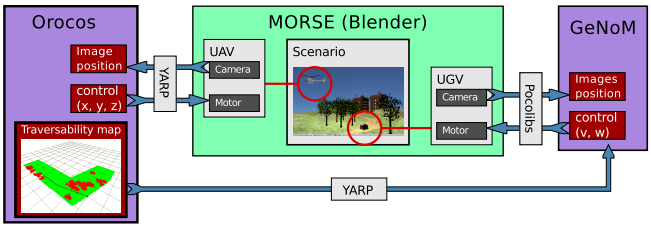


Fig. 6: Diagram of the data exchange between the different modules in the Action test scenario.

functionality of each sensor is defined within the Blender world using both the GE interface and Python scripts. As an example, the SICK laser works by using the ray tracing functions existing in Blender; the field of view of the SICK is visualised by deforming the geometry of an arc shaped mesh, using the distances measured with ray tracing.

The camera sensor can provide data with different levels of abstraction. Basic images can be generated using Blender's camera object and the `VideoTexture` library of the Game Engine. These can be used, for example, to generate Digital Terrain Maps (DTM), through stereo vision. Alternatively, a camera can produce a semantic list of the visible objects within its field of view, generated by using predefined Blender functions to locate objects specifically tagged. This permits testing higher level tasks without the burden of image processing to identify objects. Fig. 5 illustrates these two kinds of camera in use.

A. Practical use cases

1) ROSACE project: This project deals with the design and development of a group of robots capable of communicating and cooperating to accomplish a given objective in a dynamic environment. The test scenario for the project is a search and rescue mission in the event of fire in a rural area. In simulation, the robots must cooperate in locating various victims, while avoiding obstacles such as buildings, roads and fire. The MORSE scenario includes groups of 5 to 15 UGVs outfitted with thermometer sensors and simulated communication radios. The simulation is used to study the different cooperation strategies and the self adaptability of the robot team in case of loss of communication. The robot controlling agents are programmed in Java and communicate data with MORSE via YARP, serialised using JSON, and decoded and interpreted in both ends.

2) ACTION project: Related to the cooperation of heterogeneous types of robots (land, air, sea and submarine) in various localization and detection scenarii, in complex outdoor environments. In the first test scenario, an UGV requests a traversability map of a certain direction to a nearby robot helicopter (UAV). The latter flies in the direction requested, locates obstacles that could impede the movement of the ground vehicle, generates a map and sends it back. Upon receiving the map, the UGV recomputes its projected path and starts the cycle again. In simulation, both robots use completely different evaluated software, running in two separate computers. Both communicate with a single instance of MORSE, each using a distinct middleware (Fig. 6). The UGV is controlled with G^{en}oM modules and communicates with MORSE via the Pocolibs library. Meanwhile, the UAV is driven by Orocos [26], and uses YARP to talk to MORSE[1]. This scenario demonstrates the interest of the "Software-in-the-Loop" concept, and that MORSE can link to various middlewares.

3) Simulation of ReSSAC: The ReSSAC is an Unmanned Autonomous Helicopter based on a YAMAHA RMax helicopter. The objective of the project is to perform air–to–ground target tracking missions in an unknown environment [27]. The evaluated software in this application is a visual target search and tracking system implemented in Orocos. It tracks the movement of a ground vehicle and generates motion controls for the helicopter to follow the target. The ReSSAC experiment is a proof of concept for the simulator, where an existing algorithm that works in the real robot was later tested in the simulator using "Software-in-the-Loop", connecting via the YARP middleware. This example demonstrates that the simulator can be correctly connected with the evaluated software, properly emulate the output of the sensors, and interpret the control instructions for the robot. In the test simulation, the ground target is made to move with keyboard commands, while the simulated helicopter tracks and follows the vehicle, giving the same results as in the real life experiment.

V. Future work

The simulation of multiple robots in MORSE is currently limited by the processing power of the computer running the software. Up to 15 robots with simple sensors can be simulated in a single scenario with MORSE turning at over 20 frames per second. The generation and processing of image data (as in the case of simulated cameras) does require more resources, depending on camera resolution and data output rate. To cope with this problem, we plan to deploy a number of simulator nodes, each one running the same simulated scenario, but managing only a limited number of virtual robots. The different nodes need to be synchronised through a special node, denoted the *simulation Supervisor*. Each node is responsible of one or more robot,

[1]Both robots communicates through YARP, but this is related to the way multi-robots interactions have been defined, and not to MORSE.

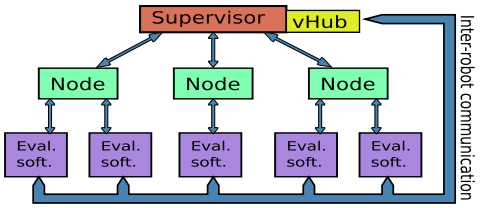


Fig. 7: Distributed architecture of MORSE, including the Supervisor, *vHub* and multiple simulation nodes.

and broadcasts to other node their positions and actions. Concerning other objects, the information goes through the *Supervisor* which is responsible of the consistency of the whole "simulation universe". Furthermore, the *Supervisor* also provides a central management for the simulation. A special GUI allows to start or stop nodes, to add or remove some robots or some objects. It can allow too to introduce hazard in the simulated environment.

Another important task in a multi-robots environment is the simulation of robot communications. The *Virtual Hub* (vHub) component, associated to the *Supervisor* node, will achieve this. Using information from the simulated world (*i.e.* the distance between robots, line of sight), the *vHub* decides if two robots can communicate, and in which conditions (packet loss, congestion, slow bandwidth ...). Moreover, the *vHub* component should log all communication flow for future analysis. Fig. 7 sums up the future extension to the MORSE architecture. The decomposition of the simulation in different nodes will allow to simulate numerous robots in a realistic way. The *vHub* will allow to experiment with the communication issues within the simulator.

VI. Summary

We have presented a new robotics simulator, completely based on Blender and Python. MORSE is designed for high reusability under all kinds of robotics research. It provides many facilities to be integrated with existing robotics software and to simulate numerous sensor and actuators setups. This simulator is already used in real projects, demonstrating it can fulfil the requirements specified of modularity, middleware independence and realistic visualisation and physics. Many features of the simulator are still in an early stage of development, including the multi–node architecture, but the integration with ROS, G^{en}oM 3 and Blender 2.5 will make MORSE simpler to use in various robotics laboratories.

MORSE is developed as an open–source project, the source code can be downloaded from the GIT repository: (`http://github.com/laas/morse.git`)

User documentation and additional information is also available at (`http://morse.openrobots.org`)

Acknowledgments: This work has been partially supported by the DGA founded Action project (`http://action.onera.fr`) and the STAE foundation Rosace project (`http://www.fondation-stae.net`)

References

[1] *Workshop on robot simulators: available software, scientific applications and future (along with IROS'08, Nice, France)*, Sept. 2008.
[2] *International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR 2008) Venice(Italy)*, Nov 2008.
[3] T. Siméon, J-P. Laumond, and F. Lamiraux. Move3d: a generic platform for path planning. In *in 4th Int. Symp. on Assembly and Task Planning*, pages 25–30, 2001.
[4] F. Kanehiro, K. Fujiwara, S. Kajita, K. Yokoi, K. Kaneko, and H. Hirukawa. Open architecture humanoid robotics platform. In *Proceedings of the IEEE ICRA*, pages 24–30, 2002.
[5] V. Zykov, P. Williams, N. Lassabe, and H. Lipson. Molecubes extended: Diversifying capabilities of open-source modular robotics. International Conference on Intelligent Robots and Systems (IROS), workshop on Self-Reconfigurable Robotic, 2008.
[6] F. Kanehiro, H. Hirukawa, and S. Kajita. Openhrp: Open architecture humanoid robotics platform. *I. J. Robotic Res.*, 23(2):155–165, 2004.
[7] O. Michel. Webots: Professional mobile robot simulation. *Journal of Advanced Robotics Systems*, 1(1):39–42, 2004.
[8] M. Lewis, J. Wang, and S. Hughes. Usarsim : Simulation for the study of human-robot interaction. *Journal of Cognitive Engineering and Decision Making*, 2007:98–120, 2007.
[9] Microsoft robotics developer studio. `http://www.microsoft.com/robotics/`.
[10] Cogmation robotics simulation. `http://www.cogmation.com/`.
[11] S. Petters, D. Thomas, M. Friedmann, and O. von Stryk. Multilevel testing of control software for teams of autonomous mobile robots. In *Simulation, Modeling, and Programming for Autonomous Robots*, volume 5325 of *Lecture Notes in Computer Science*, pages 183–194. Springer Berlin / Heidelberg, 2008.
[12] B. P. Gerkey, R. T. Vaughan, and A. Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *In Proceedings of the 11th International Conference on Advanced Robotics*, pages 317–323, 2003.
[13] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *In IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2149–2154, 2004.
[14] Blender 3D. `http://www.blender.org`.
[15] Blender for robotics interest group. `http://wiki.blender.org/index.php/Robotics`.
[16] R. Smits, T. De Laet, K. Claes, H. Bruyninckx, and J. De Schutter. itasc: a tool for multi-sensor integration in robot manipulation. In *IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems*, pages 426 –433, 2008.
[17] A. Goktogan and S. Sukkarieh. Simulation of multi-UAV missions in a real-time distributed hardware-in-the-loop simulator. In *4th International Symposium on Mechatronics and its Applications (ISMA07), Sharjah, U.A.E*, March 2007.
[18] S. Joyeux, A. Lampe, R. Alami, and S. Lacroix. Simulation in the LAAS architecture. In *Workshop on principle and practice of software development in robotics, IEEE ICRA, Barcelona (Spain)*, April 2005.
[19] G. Metta, P. Fitzpatrick, and L. Natale. YARP: yet another robot platform. *International Journal of Advanced Robotic Systems*, 3(1):43–48, 2006.
[20] ROS: Robot Operating System. `http://www.ros.org`.
[21] Pocolibs. `http://pocolibs.openrobots.org`.
[22] J. Kramer and M. Scheutz. Development environments for autonomous mobile robots: A survey. *Autonomous Robots Journal*, 22:101–132, 2007.
[23] N. Mohamed, J. Al-Jaroodi, and I. Jawhar. Middleware for robotics: A survey. In *IEEE International Conference on Robotics, Automation and Mechatronics (RAM 2008), Chengduo (China)*, pages 736–742, 2008.
[24] A. Shakhimardanov and E. Prassler. Comparative evaluation of robotic software integration systems: A case study. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, San Diego (USA)*, 2007.
[25] A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. F. Ingrand. Genom3: Building middleware-independent robotic components. In *Proceedings of the IEEE ICRA*, 2010.
[26] The Orocos Project. `http://www.orocos.org`.
[27] Y. Watanabe, C. Lesire, A. Piquereau, P. Fabiani, M. Sanfourche, and G. Le Besnerais. The onera ressac unmanned autonomous helicopter : Visual air-to-ground target tracking in an urban environment. In *American Helicopter Society 66th Annual Forum*, May 2010.